Optimizer in DL

Deep Learning (DSE316/616)

Vinod K Kurmi
Assistant Professor, DSE

Indian Institute of Science Education and Research Bhopal

Aug 27, 2022

Disclaimer

- Much of the material and slides for this lecture were borrowed from
  - Bernhard Schölkopf's MLSS 2017 lecture,
  - Tommi Jaakkola's 6.867 class,
  - CMP784: Deep Learning Fall 2021 Erkut Erdem Hacettepe University
  - Fei-Fei Li, Andrej Karpathy and Justin Johnson's CS231n class
  - Hongsheng Li's ELEG5491 class
  - Mitesh Khapra Class notes

Previous class: Activation Functions

Types of Activation Functions

Function Type	***Equation***	***Derivative***
Linear	$f(x) = ax + c$	$f'(x) = a$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH	$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
Parametric ReLU	$f(x) = \begin{cases} \alpha x \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
ELU	$f(x) = \begin{cases} \alpha(e^{x} - 1) \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$

Previous class:What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

Training a neural network, main loop:

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

simple gradient descent update
now: complicate.

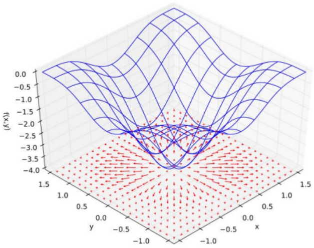
Gradients

- When we write $\nabla_W L(W)$, we mean the vector of partial derivatives wrt all coordinates of W:

$$\nabla_W L(W) = \left[\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \ldots, \frac{\partial L}{\partial W_m}\right]^T$$

  where $\dfrac{\partial L}{\partial W_i}$ measures how fast the loss changes vs. change in W_i

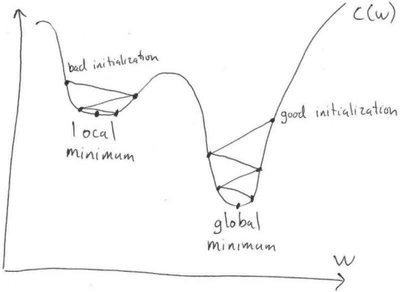
- **In figure:** loss surface is blue, gradient vectors are red:
- When $\nabla_W L(W) = 0$, it means all the partials are zero, i.e. the loss is not changing in any direction.
- Note: arrows point out from a minimum, in toward a maximum



Slide adapted from John Canny

Optimization

- Visualizing gradient descent in one dimension:



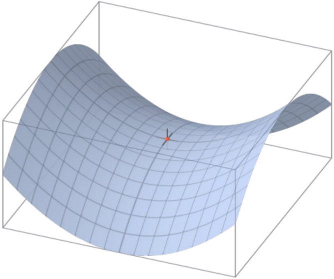
- The regions where gradient descent converges to a particular local minimum are called basins of attraction.

Local Minima

- Since the optimization problem is non-convex, it probably has local minima.
- This kept people from using neural nets for a long time, because they wanted guarantees they were getting the optimal solution.
- But are local minima really a problem?
  - Common view among practitioners: yes, there are local minima, but they're probably still pretty good.
    - Maybe your network wastes some hidden units, but then you can just make it larger.
  - It's very hard to demonstrate the existence of local minima in practice.
  - In any case, other optimization-related issues are much more important.

Saddle Points

- At a saddle point, $\frac{\partial L}{\partial W} = 0$ even though we are not at a minimum. Some directions curve upwards, and others curve downwards.
- When would saddle points be a problem?
  - If we're exactly on the saddle point, then we're stuck.
  - If we're slightly to the side, then we can get unstuck.



Batch Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Compute gradient estimate over N examples:

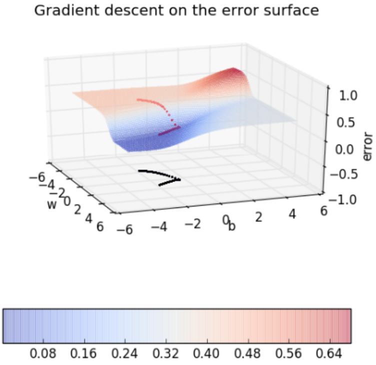
3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$

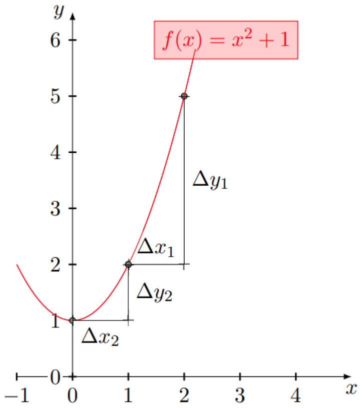
4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

5: **end while**

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

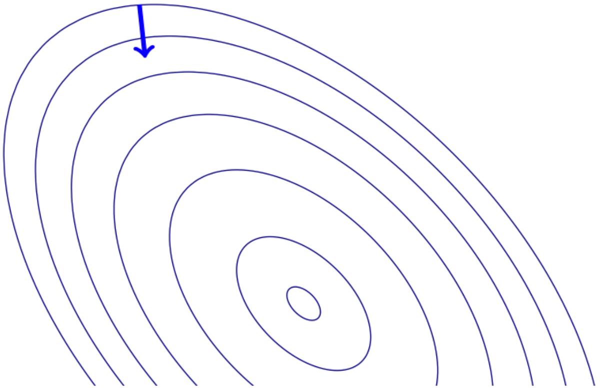
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```



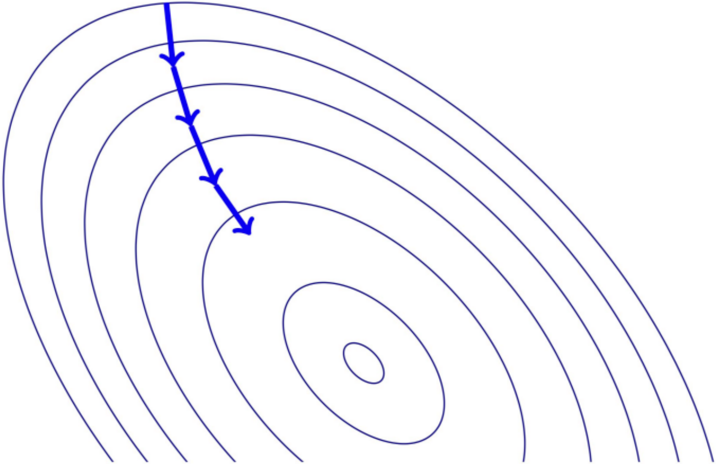


- When the curve is steep the gradient $(\frac{\Delta y_1}{\Delta x_1})$ is large
- When the curve is gentle the gradient $(\frac{\Delta y_2}{\Delta x_2})$ is small
- Recall that our weight updates are proportional to the gradient $w = w - \eta \nabla w$
- Hence in the areas where the curve is gentle the updates are small whereas in the areas where the curve is steep the updates are large

Gradient Descent



Gradient Descent



Stochastic Batch Gradient Descent

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: Apply Update: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

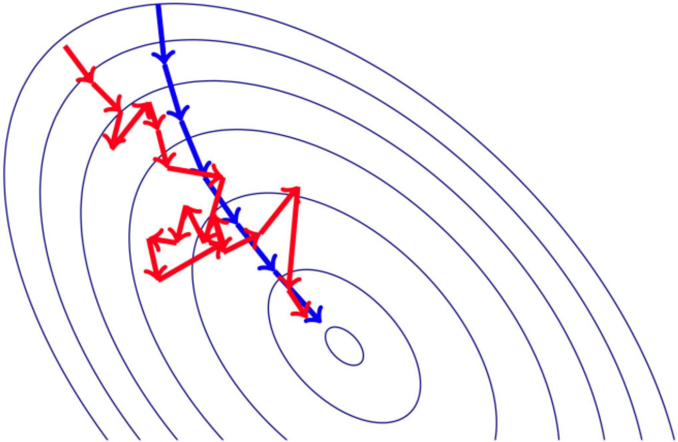
6: **end while**

Minibatching

- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- **Advantage:** Computation time per update does not depend on number of training examples N
- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

/

Stochastic Gradient Descent

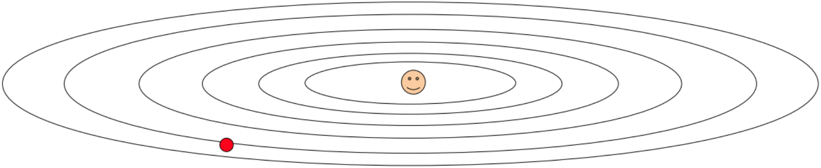


Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

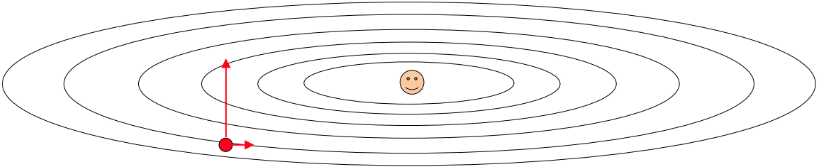
Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	$\frac{N}{B}$

Suppose loss function is steep vertically but shallow horizontally:



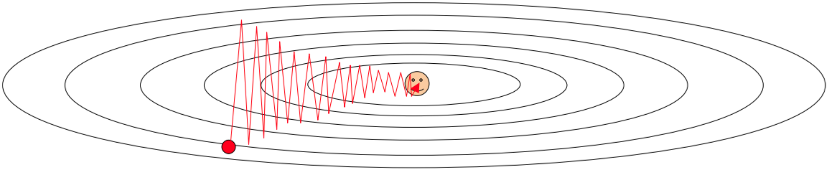
Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

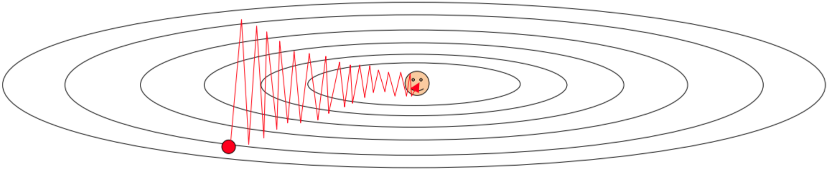
Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

very slow progress along flat direction, jitter along steep one

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

very slow progress along flat direction, jitter along steep one

Momentum based Gradient Descent

Some observations about gradient descent

- It takes a lot of time to navigate regions having a gentle slope
- This is because the gradient in these regions is very small Can we do something better ?
- Yes, let's take a look at **'Momentum based gradient descent'**

Momentum based Gradient Descent

Update rule for momentum based gradient descent

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

- In addition to the current update, also look at the history of updates.

Momentum update

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

Momentum update

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

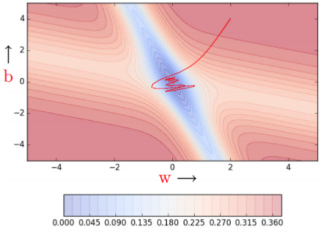
SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

- Momentum based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley
- Takes a lot of u-turns before finally converging
- Despite these u-turns it still converges faster than vanilla gradient descent
- After 100 iterations momentum based method has reached an error of 0.00001 whereas vanilla gradient descent is still stuck at an error of 0.36



Nesterov Accelerated Gradient Descent

Question

- Can we do something to reduce these oscillations ?
- Yes, let's look at Nesterov accelerated gradient

Observations about NAG

- Looking ahead helps NAG in correcting its course quicker than momentum based gradient descent Hence the oscillations are smaller and the chances of escaping the minima valley also smaller

Intuition

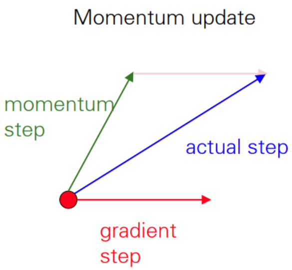
- Look before you leap
- Recall that $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$
- So we know that we are going to move by at least by $\gamma \cdot update_{t-1}$ and then a bit more by $\eta \nabla w_t$
- Why not calculate the gradient (∇w_{look_ahead}) at this partially updated value of w ($w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$) instead of calculating it using the current value w_t

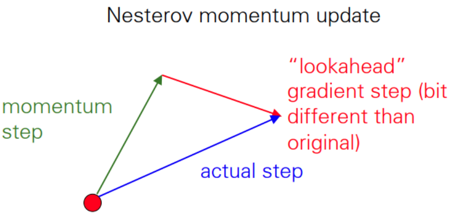
Update rule for NAG

$$
\begin{aligned}
w_{look\_ahead} &= w_t - \gamma \cdot update_{t-1} \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead} \\
w_{t+1} &= w_t - update_t
\end{aligned}
$$

We will have similar update rule for b_t

Momentum update





Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Regularization and training details

Next Class..